SYSTEM AND METHOD FOR PROVIDING SCALABLE MANAGEMENT ON COMMODITY ROUTERS

SPECIFICATION

CROSS REFERENCE TO RELATED APPLICATIONS

The present application claims priority from U.S. Provisional Patent Application Serial No. 60/461,221 filed April 7, 2003, the entire disclosure of which is incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field Of The Invention

The present invention relates to a system, method and software arrangement for providing distributed network management and, more particularly to a system, method and software arrangement for providing pattern-based distributed network management wherein lightweight mobile agents are distributed throughout a network based upon a pre-defined pattern to perform network management functions.

2. Background Art

With the increase in the amount of data exchanged over communication networks, managing the flow of data to avoid bottlenecks in networks and to ensure that a high throughput of data is achieved has become necessary in many situations. Centralized network management schemes have evolved to optimize the performance of networks. Over the last decade, certain drawbacks of existing centralized management schemes have been recognized and several approaches to realizing distributed network management have been developed. Most of the distributed network management schemes have focused on distributing the computations associated with the management task while keeping the overall control of the network management tasks centralized.

Initially, high-performance active network platforms were developed to implement network management applications that process management traffic at speeds nearing wire-speed. Systems implemented with this philosophy typically have low-level, assembly-style instruction sets, optimized for space and speed. Unfortunately, developing management programs on such systems is difficult, due to

their low-level nature, and thus the use of these management programs is limited to simple network management functions, such as programmable traffic probes. Furthermore, these platforms are built on customized or special network nodes that require features not available in off-the-shelf routers, referred to as commodity routers.

In order to apply some of the techniques used in high-performance active network platforms to the management of traditional internet protocol ("IP") networks, software routers have been used. In these environments, the operating system intercepts packets for processing in a management execution environment, usually through an IP option called Router Alert. Many of the systems developed along this line emphasize flexibility over performance and generally require the support of a heavyweight infrastructure, such as those built on Java.

A significant step towards decentralized network management has been made with the introduction of mobile agents for management tasks. Mobile agents can be characterized as self-contained programs that move through the network and act on behalf of a user (i.e., a human operator) or another entity. Mobile agents are generally complex, since they often need a degree of intelligent behavior for autonomous decision-making. The mobile agents execute various network management tasks in a distributed manner. A major drawback of the mobile agents approach to distributed network management lies in the structure of the mobile agents themselves. The mobile agents are fairly complex, requiring more processing power than is typically present in standard routers or switches and require significant bandwidth in order to propagate throughout the network.

For mobile agents to effectively propagate through the network, various graph traversal algorithms are utilized to control and coordinate the processing and aggregation of management information inside the network. From the perspective of a network manager, the algorithms provide the means to 'diffuse' or spread the computational process over a large set of nodes. A key feature of the approach is its ability to separate this mechanism of diffusion and aggregation from the semantics of the management operation. The paradigm achieves this through the development of two important concepts: the navigation pattern and the aggregator. The navigation pattern represents the generic graph traversal algorithms that

implement distributed control while the aggregator implements the computations required to realize the task.

Figs. 1A – 1D represent the simplest examples of navigation patterns. A basic pattern 100 is illustrated in Fig. 1A, where control moves (represented by arrow 106) from one node 102 to another node 104 and returns (represented by arrow 108) after triggering an operation in the node 104. A manager-agent interaction is an example of this pattern. Another pattern 120 represents the scenario where control begins at an originator node 122 moves along a number of nodes 124 in a path in the network, triggers operations on the nodes 124 of this path, and returns to the originator node 122 along the same path. A possible application of this pattern is resource reservation for a virtual path or a multi-protocol label switching ("MPLS") tunnel. Still another pattern 140 is illustrated in Fig. 1C, where control begins at an originator node 142 migrates to a node 144, then in parallel to neighboring nodes 146, 148, triggers operations on these nodes, and returns with result variables to the originator node 142. The pattern 140 can be understood as a parallel version of the pattern 100. Finally, in a further pattern 160, shown in Fig. 1D control moves along a circular path in the network. It is important to note that navigation patterns can be defined independently of the management tasks performed in an operation. A drawback of these propagation schemes is their non-determinative nature.

SUMMARY OF THE INVENTION

It is therefore one of the objects of the present invention to provide an apparatus, method and software arrangement that allows management information to be collected and processed quickly from a large network of commercial routers without the need for proprietary access to internal software or hardware of the commercial routers and without requiring any modifications to be made to internal software or hardware of the commercial routers.

It is another object of the present invention to provide an apparatus, method and software arrangement including management programs configured to exploit the speed and power associated with the parallel processing capabilities of a

network without incurring the usual complexities associated with parallel programming.

It is still another object of the present invention to provide an apparatus, method and software arrangement for extending the functionality of the management system, the system does not have to be brought down in order to introduce new code into it. (i.e. the code can be transported by the mobile agents).

It is yet another object of the present invention to provide an apparatus, method and software arrangement for dramatically lowering the cost of developing parallel management programs because a single pattern program can be reused for many applications by plugging it with different aggregators.

It is a further object of the present invention to provide an apparatus, method and software arrangement for collecting and processing information using the Echo pattern in an efficient manner both in terms of speed and bandwidth consumed for networks with a tree or scale-free topology.

These and other objects can be achieved with the exemplary embodiment of the apparatus, method and software arrangement for providing pattern-based decentralized network management according to the present invention. The exemplary apparatus is an network node including a router connected to at least one other network node of a plurality of network nodes and a processor. The processor is configured to receive a network management program and mobile state information, determine whether to send the network management program to the at least one other network node of the plurality of network nodes, transmit the mobile state information to the at least one other network node of the plurality of network nodes, and selectively transmit the network management program based upon the determination to the at least one other network node of the plurality of network nodes.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention and its advantages, reference is now made to the following description, taken in conjunction with the accompanying drawings, in which:

Figs. 1A – 1D show examples of prior art network navigation patterns;

Fig. 2 illustrates an exemplary embodiment of a pattern-based network management system according to the present invention;

Fig. 3 illustrates an exemplary embodiment of an Echo pattern of network management program dissemination according to the present invention;

Fig. 4 illustrates an exemplary embodiment of a software architecture of an active node of the pattern-based network management system according to the present invention;

Figs. 5A – 5D shows an exemplary embodiment of a process for distributing a new network management program throughout the pattern-based network management system of Fig. 2 according to the present invention;

Fig. 6 illustrates an exemplary embodiment of a process for distributing a new network management program from active node to active node throughout the pattern-based network management system of Fig. 2 according to the present invention; and

Fig. 7 shows a graph comparing the scalability of the pattern-based network management system against a centralized network management system.

Throughout the drawings, the same reference numerals and characters, unless otherwise stated, are used to denote like features, elements, components, or portions of the illustrated embodiments. Moreover, while the present invention will now be described in detail with reference to the Figs., it is done so in connection with the illustrative embodiments.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In an exemplary embodiment of the present invention, a pattern-based network management system 200 is provided for managing a network in a decentralized manner using a pattern-based paradigm. The system 200 realizes a pattern-based network management paradigm using a network of low-cost, single-board computers that are attached to commercial off-the-shelf routers, referred to as commodity routers. This system 200 utilizes the coordinated actions of lightweight mobile agents that can be deployed on existing networks. These lightweight mobile agents execute network management operations in a fast and scalable manner. The

lightweight mobile agents exploit the parallel processing capability of the network. These agents always carry state information with them, but carry a network management program only when absolutely necessary, rendering the lightweight mobile agents quite small when transmitted from node to node in the network. The lightweight mobile agents are configured to be distributed on a network of commodity routers and are designed such that no modifications to the routers are required.

The lightweight mobile agents manage the network in a distributed manner using pattern-based management. Pattern-based management centers around the concept of the navigation pattern, used for controlling and coordinating the actions of the lightweight mobile agents. Navigation patterns realize graph traversal algorithms that determine the dissemination of local management operations throughout the network and the aggregation of the results of these local operations.

The system 200 utilizes an efficient implementation of a pattern-based management system, whereby the network management programs disseminated by the lightweight mobile agents complete execution quickly, even in large networks. Also, the network management programs are designed to execute on virtually all commodity routers.

Pattern-based management systems have several advantages over other network management approaches. First, management programs can be formally analyzed with respect to performance and scalability. This analysis is based on the analysis of the graph traversal algorithm and its underlying pattern. Second, navigation patterns allow the separation of the semantics of management operation and the distributed programming aspects of the operation. From a software engineering perspective, this separation allows application and/or network developers to design generic patterns that can be combined with specific semantics to implement a particular management operation in a straight forward manner. A pattern, once designed, can be reused in the implementation of many management tasks. Conversely, a specific management task can potentially be built from a choice of patterns, which enables application and/or network developers to build management operations with different performance profiles. Ultimately, this approach frees an application programmer from developing distribution algorithms, allowing him/her to

focus on the management task at hand, by selecting a navigation pattern from a catalogue that captures the requirements for that task.

Third, the response to network faults, which can be complex to understand and handle, can be programmed into a pattern, thereby eliminating the need for the application programmer to deal with faults. Finally, the degree of code mobility can be controlled in a fine-grained manner, since the execution of a management operation in a network involves distributing only those parts of the program which are not already resident in the network nodes. In other words, for a management program that is frequently executed, only the states of the distributed computation need to be exchanged between network nodes, not the code, which is locally available.

Various management operations can be realized in the system 200 through the network management programs of the lightweight agents. The lightweight agents are distributed throughout the system 200 based on an Echo pattern 300 (shown in Fig. 3). These various management operations are useful in managing a network having a very large number of nodes, a dynamic topology and no centralized topology database. All management computations are performed on logical network nodes, i.e., inside the network. The management operations include statistical abstractions of the topology, statistical abstractions of local performance data, correlation of performance data with topological data, and software configuration management. Statistical abstractions of the topology allows the network management station 202 to discover various data about the network including: the current local network topology about a given node, the number of leaf nodes in the network topology, the diameter of the network, the connectivity distribution of the network topology, and the like. Statistical abstractions of local performance data allows the network management station 202 to determine the most congested links in the network, and/or the distribution of the link load, queue lengths, and the like in the network. Correlation of performance data with topological data allows the network management station 202 to identify sub-topologies where all links are highly loaded, compute the routes and the traffic volume that flows between two sub-regions of the network within a certain time period, and the like. Software

configuration management allows the network management station 202 to install a software patch on all routers running a particular version of an operating system.

A concern relating to the network management system 200 is code mobility. Specifically, determining whether code should be transferred from logical network node to logical network node in its binary or source code form, and whether code is to be transported by patterns or downloaded from code servers. The decisions made in addressing these questions have implications on the performance of the system 200 and its vulnerability to attacks.

If a program is to be transferred in source form, then a time-consuming compilation process must be invoked prior to its execution on each active node. In addition, every node must be equipped with sufficient disk space to store the compiler, the linker, and the header files. The advantage of transferring source code is that source programs are significantly smaller than compiled programs. The system 200 transfers the network management programs in binary form from logical network node to logical network node throughout the network using patterns.

In an exemplary embodiment of the present invention, code servers may be used disseminate the network management program. As the network becomes larger, however these code servers become bottlenecks. Increasing the number of code servers to overcome this issue introduces other problems, because keeping copies of the network management program on all code servers throughout the network up-to-date and consistent can be expensive and time consuming.

In an exemplary embodiment of the present invention, the system 200 transfers the network management program in its source code form. In another exemplary embodiment of the present invention, the system 200 transfers the network management program in its source code form to a first logical network node only, which compiles the network management program into its executable code form, which is then transmitted throughout the network.

As with any system that includes mobile code, there is always a danger of unsafe or malicious programs being introduced. The system 200 attempts to address this issue in several ways. First, all communication between a network management station 202 and a logical network node occurs through an secure socket layer enabled ("SSL-enabled") web interface. This reduces the risk of unauthorized

access and protects against masquerade attacks. Second, the compiled code or the executable of a management program is only executed within the context of a separate process with restricted rights and resource quotas. This prevents management programs from interfering with one another or from crashing the daemon, should a fatal error occur. Finally, the communication channel between each of the logical network nodes 220, 230, 240 are implemented using a simple protocol, similar to transport layer security ("TLS") protocol or the like, thereby preventing a third party from altering the program code or state while the data is in transit. In addition, all services that come with the operating system running on the logical network nodes 220, 230, 240 and are not used by the system 200 are disabled.

Fig. 2 illustrates a network architecture of the pattern-based network management system 200 according to an exemplary embodiment of the present invention. The system 200 includes the network management station 202, a code server 204 and a plurality of logical network nodes 220, 230, 240. Each of the plurality of network nodes 220, 230, 240 include a physical router 208, 210, 214 in communication with an active node 206, 212, 216, respectively. Copies of a pattern-based network management program run in the active nodes 206, 212, 216 of the network nodes 220, 230, 240. The pattern-based management program can be realized in a number of ways, for example, using a general purpose framework, such as Java, or an active networking toolkit, such as ANTS, which is described in more detail by http://tns-www.lcs.mit.edu/publications/openarch98.html, incorporated herein in its entirety by reference. Execution of the pattern-based network management program begins when a lightweight mobile agent containing the network management system is transmitted from the network management station 202 to the active node 206 of the first network node 220 (sometimes called the start node or originator node). When the program has completed its execution cycle on the first node 220, a pattern component of the program determines the appropriate subsequent node or nodes on which the program must execute next. If the subsequent node already contains a copy of the pattern-based network management program, the lightweight mobile agent is transmitted containing only the program's state information to the subsequent node. Otherwise, the lightweight mobile agent is sent

containing the network management program in its executable form and mobile state variables.

Each router 208, 210, 214 is managed by a dedicated active node 206, 212, 216 that hosts the execution environment needed for running the pattern-based network management program. Each active node 208, 210, 214 includes an internet-enabled, single-board computer including a network interface and a storage device. The hardware of the active node 206, 212, 216 is commercially available in the form of an aluminum cube of 3 inches per side. Each active node 206, 212, 216 runs a modified distribution of the Unix operating system having limited functionality.

In an exemplary embodiment, the internet-enabled, single-board computer is equipped with an Intel StrongARM 1110 microprocessor, 32MB of SDRAM, and a 10Mbps Ethernet interface. In another exemplary embodiment, the storage device is a 1 GB drive connected to the onboard compact flash slot of the single-board computer. In a further exemplary embodiment, each of the active nodes 206, 212, 216 runs a Linux kernel, as well as, an Apache web server, which is used to implement the wide area network's management interface.

In an exemplary embodiment of the present invention, the active nodes 206, 212, 216 of the network nodes 220, 230, 240 may be included within the routers 208, 210, 214, respectively. Realizing such a system at low cost on commodity routers considerably restricts the design space and poses a significant challenge to network and/or application designers. In another exemplary embodiment of the present invention, the network management station 202 executes the actions of the code server 204, thereby simplifying the hardware used in the system 200.

From the perspective of scalability, a pattern-based management system can eliminate bottlenecks associated with centralized processing of management data, by distributing load to network nodes via an appropriate navigation pattern. By using an Echo pattern 300, shown in Fig. 3, as a way to distribute computation to network nodes, highly scalable management programs can be implemented in compact form.

Several navigation patterns are described above in connection with Figs. 1A – 1D. The Echo pattern 300 is an extension of the basic pattern 140 and is particularly efficient at distributing and aggregating data over large networks. The

Echo pattern 300 is based on a class of distributed graph traversal algorithms known as wave algorithms. The Echo pattern 300 represents an efficient and elegant scheme for enabling global management operations in complex and large networks. The Echo pattern 300 dynamically adapts to changes in the network topology, does not require up to date network information, and scales well in very large networks. The time complexity of the Echo pattern 300 increases linearly with the network diameter, which results in fast execution times in networks with a connectivity distribution that follows the power law. The traffic complexity of the Echo pattern 300 grows linearly with the number of network edges, and the management traffic produced by executing this pattern is distributed evenly across all links, without causing hot spots, where congestion can occur.

The behavior of the Echo pattern 300 can be described as follows. The pattern 300 starts out from an originator node 302, migrating to each of its neighbors 304, 306 for further execution during an expansion phase of the Echo pattern 300. During the expansion phase, a lightweight mobile agent propagates (represented by arrows 312, 314, 316, 318) from one node to each neighbor node. A lightweight mobile agent, arriving on a node for the first time, marks the node as 'visited' (for example, node 304) and transmits a lightweight mobile agent to each neighbor (here nodes 308, 310), except for the one from which it arrived, which is called its parent (here node 302). Lightweight mobile agents arriving on a node that has been marked as 'visited', terminate at that node (i.e., they do create more explorers). If the node has no neighbors other than its parent (for example, nodes 306, 308, 310), the lightweight mobile agent returns to its parent node. This return of the pattern from a node to its parent is called an echo. When a node has received an echo from each of its neighbors, it returns an echo to its parent. The Echo pattern terminates when the originator node 302 has received an echo from each of its neighbors.

Fig. 4 illustrates a software architecture 400 of the active nodes 206, 212, 216 of the network management system 200. The software architecture 400 of each of the active nodes 206, 212, 216 are essentially the same and for the sake of simplicity, the software architecture 400 will be described in terms of the active node 206. The software architecture 400 includes an active node manager 402, a compiler 416, an active node engine 404 and a group of data repositories 414, 418, 420, 422,

424. The active node manager 402 offers a web interface to the network management station 202 for configuring and operating the active node 206. The active node engine 404 includes a preprocessor 406, an execution environment 408, a transport access point 410 and a device manager 412. The active node engine 404 runs as a background process. It implements the execution environment 408, which runs the pattern-based network management program on the active node 206. The node state repository 414 stores the operational state of the active node 206 including the numbers and parameters of executing management programs. The source repository 416 stores source code associated with a number of programs. The binaries repository 418 stores a cache of ready-to-run patterns and aggregators. The local program state repository 422 stores local state variables when a lightweight mobile agent migrates to another node. The management operation result repository 420 provides for persistent storage of results returned from a management operation.

In an exemplary embodiment, the active node manager 402 includes an Apache SSL-enabled web server and a set of server-side scripts, such as hyper text preprecessor ("PHP") scripts.

Figs. 5A – 5D illustrate a process 500 for distributing a new network management program on the system 100. The process 500 begins at step 502 when the active node manager 402 of the start node, here the active node 206, receives a lightweight mobile agent containing a network management program in its source code from the network management station 202. The start node (sometimes referred to as the originator node) is simply the first active node to receive the new source code. Once the active node manager 402 receives the complete transmission, the active node manager 402 stores the new source code in the source repository 414 and relays the source file names and the parameters to the preprocessor 406 at step 504.

At step 506, the preprocessor module 406 invokes the compiler 416 to process the source code of the new portion of the new network program and computes a digital fingerprint, such as, an MD5 checksum of the resulting binary, to verify the integrity of the data. At step 508, the compiler determines whether an error occurred during compilation. If the compiler 416 encountered an error while processing, the compilation is aborted and the process 500 advances to step 509. Otherwise, the process 500 advances to step 510 and the preprocessor 406 invokes the execution

environment 408 to run the new network management program. At step 512, the process 500 dynamically loads the program and instantiates the pattern and aggregator objects. At step 509, the execution environment 408 returns an error code to the network management station 202 via the active node manager 402.

At step 514, the process 500 determines whether program binaries are in the binary repository 418. If the program binaries are already in the binary repository 418, the preprocessor 406 invokes the execution environment 408 directly at step 516, passing to it the filenames and paths of the compiled binaries. Otherwise, the process 500 advances to step 517, where the execution environment 408 returns an error code to the network management stations 202 via the active node manager 402 and the process 500 exits.

At step 518, the execution environment 408 dynamically loads the new network management program, the pattern object and the aggregator object and begins execution of the new network management program. The execution environment 408 also generates a system-wide unique cookie at step 520, which associates the distributed state of the program with its current execution. The execution environment 408 relinquishes control to the device manager 412 at step 520, passing the arguments as specified by the network management station 202 to the device manager 412 for further processing.

At step 524, the network management program accesses the management interface of the attached router through the device manager 412 in order to acquire a list of active node addresses. In addition to the specific access protocol, the device manager 412 implements low-level monitoring procedures, such as heartbeats, to detect failures in attached devices. The active node 206 includes a single device manager 412 for a simple network management protocol ("SNMP") protocol. When the network management program has completed its execution on the active node 206, it returns control and a list of active node addresses to the execution environment 408 at step 526. The list of active node addresses are the active nodes the lightweight mobile agent will migrate to next.

In an exemplary embodiment of the present invention, the active node 206 includes multiple device managers, one for each access protocol supported by the router 208.

At step 528, the execution environment 408 concludes its active execution cycle. The execution environment 408 stores the local program variables in the local state repository and serializes the mobile state variables. Once the mobile state variables are serialized, they are passed to the transport access point 410, along with the list of node addresses, a request to transmit data to those addresses and the cookie.

The main function of the transport access point 410 is to securely transfer program code and states between adjacent nodes. Whenever an active node is initialized, it connects to its neighboring active nodes by establishing secure channels. When a transport access point 410 receives a request to send a lightweight mobile agent to a list of active node addresses, the transport access point 410 determines whether any additional addresses exist in the list of active node addresses at step 530. If no additional addresses exist, the process 500 exits.

Otherwise at step 532, the transport access point 410 determines whether the active node indicated by the current address of the list of addresses has a copy of the current executable. The transport access point 410 makes this determination by reading records of an executable transmission table. Each record of the executable transmission table contains an active node address and the digital fingerprint, such as, an MD5 program checksum, for each executable sent or received by the transport access point 410. If no record exists wherein an active node address of a particular record matches the current address and the digital fingerprint of the particular record matches the digital fingerprint associated with the current executable, the executable is sent to the node associated with the current address, despite the chance that the neighbor might actually have a copy of the executable. While this scheme incurs a slight overhead, it is simple and requires no handshake between the active node 206 and neighboring active nodes. If an active node has the executable, the process 500 advances to step 536 and the transfer access point 410 transmits the mobile state variables and the cookie to the active node at the current active node address. Otherwise at step 538, the transfer access point 410 transmits the mobile state variables, the cookie and the executable code to the active node at the current active node address. The transfer access point 410 also updates its records

reflecting the transmission of the executable having a particular digital fingerprint to the current address.

In order to distribute the new network management program throughout the system 100, various active nodes will send the new network management program in its executable form to other active nodes. Fig. 6 illustrates a process 600 for distributing the new network management program from active node to active node throughout the system 100. The process 600 begins when a lightweight mobile agent containing the executable is received by the transport access point 410 at step 602. At step 604, the transport access point 410 stores a record in the executable transmission table reflecting the active node address from which the lightweight mobile agent came and the digital fingerprint of the executable form of the new network management program. Once the record is written, the transport access point 410 stores the executable in the binary repository 418 at step 606. At step 608, the transport access point 410 invokes the execution environment 408 to run the new network management program.

Once the execution environment 408 is running, the process 600 must determine whether the executable has run on the current active node. Using the system-wide cookie, the execution environment of the current active node determines whether the node has participated in the current execution of the program. When the program is first executed on a node, the system-wide cookie is stored on the node by the execution environment. If the program returns to the node at some later time, the execution environment can check to see if it has already stored the cookie. If the system-wide cookie associated with the program is already stored on the node, the node has already participated in the current execution of the program. Otherwise, it is the first time the node is executing the program. When the program completes its final execution on the node, its associated cookie is deleted by the execution environment. If the executable has executed on the current node, there is no need to instantiate the pattern and aggregator objects again, so the process 600 advances to step 614. Otherwise at step 612, the process 600 dynamically loads the executable and instantiates the pattern and aggregator objects.

At step 614, the process 600 reads the local state variables of the new network program from the local program states repository 422. The execution

environment 408 dynamically loads the pattern and aggregator objects and executes the new network management program at step 616. After the execution of the new network management program begins at step 616, the process 600 passes control to the process 500 at step 522 for the remainder of its execution.

In order to test the efficacy of the pattern-based network management system 200 and develop pattern-based management programs, a discrete-event simulator was developed. The discrete-event simulator is capable of simulating a large pattern-based management system of up to 60,000 nodes. Pattern-based management programs are loaded into the simulator on the fly for simulation. The simulator's interactive features allow the dynamics of a pattern to be visualized and recorded when its associated management program is executed. Performance data, such as completion time and volume of management traffic generated, can be collected and analyzed in order to determine the efficacy of management programs and improve thereon.

In an exemplary embodiment, the discrete-event simulator is a C++ application executing on a personal computer running a Microsoft Windows operating system.

In order to evaluate the performance of the system 200, the execution time of a management operation is measured and calculated. Execution time is defined as the time period from when a network management program is launched, i.e. downloaded for execution, on a start node to when the results are returned to the network management station 202. Two series of experiments have been conducted to obtain a delay profile for the network management programs used on the system 200. The first series of experiments focus on measuring the delay incurred by a management program based on the pattern 100 (shown in Fig. 1A). The pattern 100 essentially models a simple polling operation, where control passes from a node to its neighbor before returning. The patterns 120, 140 can be expressed as serial compositions of patterns 100. In a similar manner, the pattern 130 is the basic building block of a class of patterns in which control is passed to neighboring nodes in parallel. Thus, the measurements obtained from benchmarking the pattern 130 will allow estimation of the performance of management programs that are based on patterns derived from the pattern 130 (one of which is the Echo pattern 300).

For accurate measurements, the experiments have been carried out on an isolated test bed of four Cisco 2621 routers, which are interconnected via a Cisco Catalyst 2900 fast Ethernet switch. Each router is equipped with 2 fast Ethernet ports, one of which is connected directly to an access node 206. A 1.13 GHz DELL Inspiron 8100 notebook serves as the network management station 202. Static routes have been setup from each router to the fast Ethernet switch, so that all nodes are able to communicate with each other.

In order to understand the delay profile of a network management program using a pattern 100, the execution of the network management program is decomposed into a series of phases, as shown in Table 1.

To measure delay from the point of view of the network management program, the first phase begins when control is passed to the network management program (T1). This is called the execution phase. When the network management program completes its execution, the serialization phase (T2) is invoked to serialize the program's mobile state variables into a buffer, which is then transmitted to the remote active node in the dispatch phase (T3). The receiving phase (T4) begins, when the mobile state variables have been received on the remote active node. The network management program code (if sent) is also saved during this phase. Depending on whether the management program has been executed on the node before, the next phase can be either the loading phase (T5) or the instantiation phase (T6).

The loading phase (T5) occurs the first time a new network management program is executed on an active node. Typical tasks performed include invoking the dynamic linker to load the program code as a shared library and instantiating the pattern and aggregator objects. If the network management program has already been loaded, due to a previous execution, the instantiation phase (T6) is performed. During the instantiation phase the pattern and aggregator objects are instantiated. There is no need to invoke the dynamic linker during the instantiation phase (T6) as is required during the loading phase (T5)

If the network management program is still active on a node (meaning that the pattern is certain to traverse the node again after management information is collected by downstream nodes), the pattern and aggregator objects are not destroyed when the program migrates to another node. In such cases, the only task performed is

a lookup to return their object references. This is called the resolving phase (T8). Finally the de-serialization phase (T7) recreates the mobile state variables in the program addresses space prior to execution. The network management program requires the mobile state variables to complete execution on a particular active node.

Table 1. Overhead incurred by each phase of execution of the Type 1 pattern

Phase	Duration in ms	Performed by Module
Execution (T1)	1.57 ($\sigma = 0.48$)	Execution Environment
Serialization (T2)	3.46 ($\sigma = 0.71$)	Execution Environment
Dispatch (T3)	1.67 ($\sigma = 0.49$)	Transport Access Point
Receiving (T4)	0.62 ($\sigma = 0.30$)	Transport Access Point
Loading (T5)	23.42 ($\sigma = 0.70$)	Execution Environment
Instantiation (T6)	0.77 ($\sigma = 0.015$)	Execution Environment
De-serialization (T7)	2.04 ($\sigma = 0.49$)	Execution Environment
Resolving (T8)	0.15 ($\sigma = 0.001$)	Execution Environment
Communications Delay (T_c)	4.04 ($\sigma = 0.10$)	---

Table 1 gives the mean and standard deviation of the delay for each of the phases (T1 through T8 and T_c), as measured over 40 runs. The communication delay on the last row of the table includes transmission delay, propagation delay and operating system overhead. The size of the mobile state variables communicated between active nodes is 207 bytes. The pattern program contains the minimal code necessary to implement the pattern 100 and does not perform any other computations. The aggregator program contains only empty functions.

The (average) completion time of a pattern 100 is derived as:

$$T_{\text{pattern 100}} = 3T1 + 2(T2 + T3 + T4 + T7) + T6 + T8 + T_c$$

For a more detailed explanation of the above formula, see K.S. Lim et. al, "Weaver: Realizing a Scalable Management Paradigm on Commodity Routers," KTH/IMIT/LCD Technical Report Nr. 02-5021, August 2002, incorporated herein by reference in its entirety. When the pattern 100 is executed for the first time, an additional delay of T5 - T6 incurs, because the execution environment 408 needs to invoke the dynamic linker. Also, the estimate given by the above equation does not take into account the situation when the node daemon is swapped out by the operating system.

Following the above approach, it is possible to derive similar expressions for the average completion times of management programs based on the patterns 120, 130, 140. Table 2 compares the estimated completion time (based on the above formula and table 1) with the actual measurements on the test bed for all the basic patterns 100, 120, 130, 140. As can be seen, the estimations lie below the measured delays in all cases, with a margin of error between 8.3% and 10%.

Table 2. Comparison of estimated vs. actual measurements for the four basic patterns

Pattern Type	Average completion time estimated)	Average completion time (measured on test bed)
Pattern 100	25.2 ms	27.6 ms
Pattern 120	72.6 ms	78.4 ms
Pattern 130	44.3 ms	47.6 ms
Pattern 140	49.5 ms	55.0 ms

In order to evaluate the scalability of the system 200 when the network to be managed becomes large, the completion times of a network management operation are measured and calculated. Specifically, the completion times of pattern-based management programs on large networks are estimated using a software program capable of simulating the execution of pattern programs and aggregators on a single computer, such as SIMPSON, which is described in more detail by K.S. Lim and R. Stadler, "Developing pattern-based management programs," Proceedings of IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS 2001), Chicago, IL, Oct. 29 2001, pp. 345-358, and encorporated herein in its entirety by reference, and the delay profiles shown above. For comparison purposes, the time of the same operation executed on a centrally management network polling nodes serially or in parallel are also estimated.

For the sake of simplicity, the network topology of the system 200 is assumed to be a full binary tree with height h. Each node in the network is a router with b+1 ports, one of which is connected to an active node. This way, each active node manages exactly one router. The latency between two adjacent routers is assumed to be 2.022ms, i.e. $T_{C1} = T_{C2} = 0.5T_C = 2.022$ms. Routes taken by packets are also assumed to be symmetrical; that is, protocol data units ("PDUs") of an SNMP request take the same path, from the manager to the managed device and vice versa. The management station is assumed to be attached to the root of the tree of the system

100 and that all management program executions use the root as the start node. Finally, the management task is selected to be an operation that computes the average value of a specific management information base ("MIB") variable across all nodes in the network.

Given the above topology, the total number of nodes in the network, N, is therefore given by

$$N = \frac{n^{h+1} - 1}{n - 1}$$

The most scalable manner to implement a task in a centralized management network solution is to compute the network-wide average value of the desired variable incrementally from the values obtained from each node that is polled via an SNMP GET. If polling is performed serially on N nodes (i.e., each GET operation must complete before the next GET is initiated), and if the time needed for the simple averaging computation is negligible, the total completion time is given by:

$$T_{centralized\_S} = (T_C + T_S) + (2T_C + T_S)n + \ldots + ((h+1)T_C + T_S)b^h$$

which evaluates to:

$$T_{centralized\_S} = \frac{(T_C(b+1) + T_S)b^{h+2} - (T_C(h+2) + T_S)b^{h+1} + T_C + T_S(1-b)}{(b-1)^2}$$

where T_S is the time required by the SNMP agent on a node to process a request. Measurements on the Cisco 2621 routers shows this to be approximately 1.9 ms. On the other hand, if the polling is performed in parallel, i.e., the system does not wait for the completion of a GET before polling the next node, and nodes farther away are polled first before nearer nodes, the total completion time is given by:

$$T_{centralize\_P} = 2hT_C + (b^h - 1)T_P$$

where T_P is the polling interval between nodes. Measurements on the active nodes of the system 200 indicate that T_P is approximately 1.5 ms.

In the case of a pattern-based distributed network management solution, the Echo pattern 300 is used to accomplish the same task as accomplished above using the centralized network management solution. The performance of the

system 100 is measured against centralized management using the serial polling scheme as a common task. Specifically, the scalability measure S is defined to be the ratio between the average completion time of a management task using the serial polling scheme and the scheme underlying the specific management task (such as parallel polling or the Echo pattern 300). Fig. 7 illustrates a chart 700 plotting the scalability measure S on the vertical axis against height of the network tree h on the horizontal axis. In each estimation the number of children of each node b is set. Line 702 shows the scalability for the Echo pattern 300 when b is set at 2. Line 704 shows the scalability for the Echo pattern 300 when b is set at 6. Line 706 shows the scalability for the parallel polling approach when b is set at 2. Line 708 shows the scalability for the parallel polling approach when b is set at 6.

From the chart 700 it is evident that parallel polling approach (using the Echo pattern 300) outperforms serial polling, since its scalability measure S never falls below 1. Furthermore, for networks of small to moderate size (i.e., b=2, h<1and also b=6, h<3), it also outperforms the Echo pattern 300 because of its lower overhead. However, for large networks, i.e., (b=6, h>4) the Echo pattern 300 yields completion times that are several orders of magnitude lower than the other schemes.

The foregoing merely illustrates the principles of the invention. Various modifications and alterations to the described embodiments will be apparent to those skilled in the art in view of the teachings herein. It will thus be appreciated that those skilled in the art will be able to devise numerous techniques which, although not explicitly described herein, embody the principles of the invention and are thus within the spirit and scope of the invention.